

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 306

January, 1988

A Proposal For An Intelligent Debugging Assistant

Ron I. Kuper

Abstract

There are many ways to find bugs in programs. For example, observed input and output values can be compared to predicted values. An execution trace can be examined to locate errors in control flow. The utility of these and other strategies depends on the quality of the specifications available. The Debugging Assistant chooses the most appropriate debugging strategy based on the specification information available and the context of the bug. Particular attention has been given to applying techniques from the domain of hardware troubleshooting to the domain of software debugging. This has revealed two important differences between the two domains: (1) Unlike circuits, programs rarely come with complete specifications of their behavior, and (2) Unlike circuits, the cost of probing inputs and outputs of programs is low.

Copyright © 1988 Massachusetts Institute of Technology

A.I. Laboratory Working Papers are produced for internal circulation and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be considered papers to which reference can be made in the literature.

Table of Contents

1 Introduction	1
2 Hardware Troubleshooting vs. Software Debugging	2
2.1 Overview	2
2.2 The Examples	3
2.3 The Role of Specifications	6
2.4 The Cost of Internal Probing	7
2.5 Finding Suspects	7
2.6 Exonerating Provably Innocent Suspects	8
2.7 Convicting a Suspect	11
2.8 Finding Suspects from Multiple Tests or Multiple Faults	12
3 The Debugging Assistant	13
3.1 Overview	13
3.2 Outline of the Debugging Algorithm	13
4 Related Work	15
4.1 Overview	15
4.2 Tutoring Systems	16
4.3 Debugging Systems	18
4.4 Other Work	21

1 Introduction

This proposal consists of 4 parts. Section 1 introduces the goals and purposes for the research. Section 2 examines the similarities and differences between the domains of software debugging and hardware troubleshooting. Section 3 describes the proposed debugging assistant. Section 4 discusses other work related to this research.

The understanding of human problem solving is a goal at the foundations of Artificial Intelligence. The Programmer's Apprentice project, of which the Intelligent Debugging Assistant is a part, is working towards this understanding by studying the domain of programming. We believe that the techniques that people use in designing and implementing programs are representative of other problem solving tasks.

Debugging is not just for novice programmers. Even experienced programmers spend much of their time correcting mistakes. Debugging is not just for newly written programs. Maintaining existing software systems would be impossible without programmers who can understand and correct one another's code.

Programmers use a variety of strategies to debug programs. They scrutinize intermediate steps of a program's execution to localize the first appearance of a bug's symptom. To detect nontermination, programmers limit the space and time resources of suspect components of the system. To confirm the presence of a bug, they look for conflicts between observed input/output values and specifications.

Since our goal is the broad understanding of problem solving, we would like to demonstrate how the same problem solving techniques are appropriate in several domains. Hardware troubleshooting is a task that appears in many ways similar to software debugging.

Hardware troubleshooters use other strategies for troubleshooting circuits. From the wiring of components in a circuit and observed misbehaviors, they form a list of devices which may be broken. By reasoning about device behavior and conservative probing of the circuit they logically eliminate devices from this list. By looking within a faulty device, they localize the fault to one of the device's subcomponents.

We will explore how principles from hardware troubleshooting can be applied to software debugging. As a practical result we will introduce new strategies for software debugging. We will also elucidate the important similarities and differences between the two domains, relative to debugging.

The proposed **Debugging Assistant** will help a programmer find errors by reasoning from first principles. In addition to using traditional software debugging techniques, the debugger will borrow strategies from the domain of hardware troubleshooting. It will choose the most appropriate debugging strategy based on the amount of specification information available and the context of the bug. The debugger will use a reasoning system [15] to perform the inference and simulation necessary for the task.

2 Hardware Troubleshooting vs. Software Debugging

2.1 Overview

The task of hardware troubleshooting appears in many ways similar to the task of software debugging. Specifically, we propose to explore the degree to which techniques that are useful for localizing faults in digital circuits are appropriate for localizing bugs in programs.

The following simplifying assumptions have been made to clarify the comparison between the hardware and software domains. These same assumptions are common to most current research on debugging systems.

- Debugging is done exclusively by reasoning from first principles, as opposed to reasoning from past experience. In other words, all reasoning is done from knowledge of structure and behavior, as opposed to having a library of “bug patterns.” A good debugging system would include both kinds of reasoning; examples of such systems are presented in Chapter 4.
- The *single-fault assumption*: incorrect program or circuit behavior is caused by exactly one bug. (We will show that this assumption may not simplify the task of software debugging at all).
- All bugs are consistent and reproducible. Due to the iterative nature of our debugging algorithm, we must be able to repeatedly reproduce error conditions.
- Programs are purely functional: they have no internal state or side effects. (We allow limited side effects in software in the form of variable assignments (i.e., LISP `setq`)).

Circuits and programs share many structural similarities. Section 2.2 will illustrate these similarities via examples.

Differences between troubleshooting and debugging can be attributed to two differences between hardware and software: (1) Unlike circuits, programs rarely come with complete specifications of their behavior, and (2) Unlike circuits, the cost of probing internal inputs and outputs of programs is low. Sections 2.3 and 2.4 will explore each difference in turn.

The logical steps taken to debug a program will parallel the steps traditionally taken to troubleshoot a circuit. Specifically, these steps are:

1. Determine an initial set of suspects.
2. Exonerate any of these suspects that can be proven innocent.
3. Convict one of the remaining suspects.

Sections 2.5, 2.6 and 2.7 will discuss each step in more detail.

Section 2.8 will address the issue of probing in the presence of multiple faults or multiple tests.

2.2 The Examples

To better illustrate the parallels between hardware troubleshooting and software debugging we will first provide one example representative of each domain.

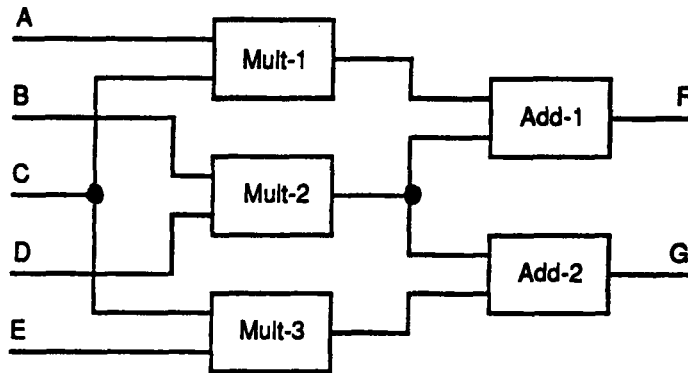


Figure 1: Adder-Multiplier Circuit

For hardware, consider the Adder-Multiplier circuit [4], shown in Figure 1. That circuit takes 5 n -bit input values, A , B , C , D , and E , and produces two n -bit output values, $AC+BD$ and $EC+BD$. In the circuit, **Mult** boxes compute the n -bit product of their two n -bit inputs, and **Add** boxes compute the n -bit sum of their two n -bit inputs.

For software, consider the trivial Air Traffic Control program (and its corresponding plan diagram [12]), shown in Figure 2. This program takes a complex data object as its input, **the-plane**, which contains information about the present state of an aircraft. The program produces as its output a new aircraft data object, representing changes to the aircraft's current position and velocity. In the program, **flight-no-of**, **position-of** and **velocity-of** are accessor functions for the aircraft data object, returning the aircraft's flight number, position and velocity vector, respectively; **build-new-plane-record** is the constructor function for the data object; **update-position** computes a new position based on the old position and velocity vector; **may-collide?** determines if the plane's present flight path may result in a collision with another plane; **wind-adjust** changes a velocity vector to account for present wind conditions; and **turn-to-safety** changes a velocity vector to avoid collisions with other aircraft.

The Adder-Multiplier circuit and the Air Traffic Control program are similar in two important ways. The first is the similarity of their representation as constraint networks [26, 28]. Schematic diagrams and plan diagrams both consist of components (shown as boxes) that are modeled by constraints on their inputs and outputs. Components are connected to one another, by wires in a circuit and by data and control flow arcs in a program.

The second similarity is that both the circuit and the program consist of components that are functionally decomposable into subcomponents (see Figures 3 and 4). Repeatedly decomposing components reveals the hierarchical structure of a circuit or program. Levels in this hierarchy correspond to layers of abstraction in the design.

There is also a significant difference between these two examples that characterizes the difference between a "software component" and a "hardware component." In hardware, the presence of multiple components of a given type, such as 3 **Mult** boxes, generally implies that each one is a separate physical instantiation. Thus we can assume that one **Mult** box is broken

```

(defun air-traffic-control (the-plane)
  (let ((flight-no (flight-no-of the-plane))
        (position (position-of the-plane))
        (velocity (velocity-of the-plane)))
    (build-new-plane-record
     flight-no
     (update-position position velocity)
     (if (may-collide? position velocity)
         (wind-adjust position (turn-to-safety position velocity))
         (wind-adjust position velocity))))))

```

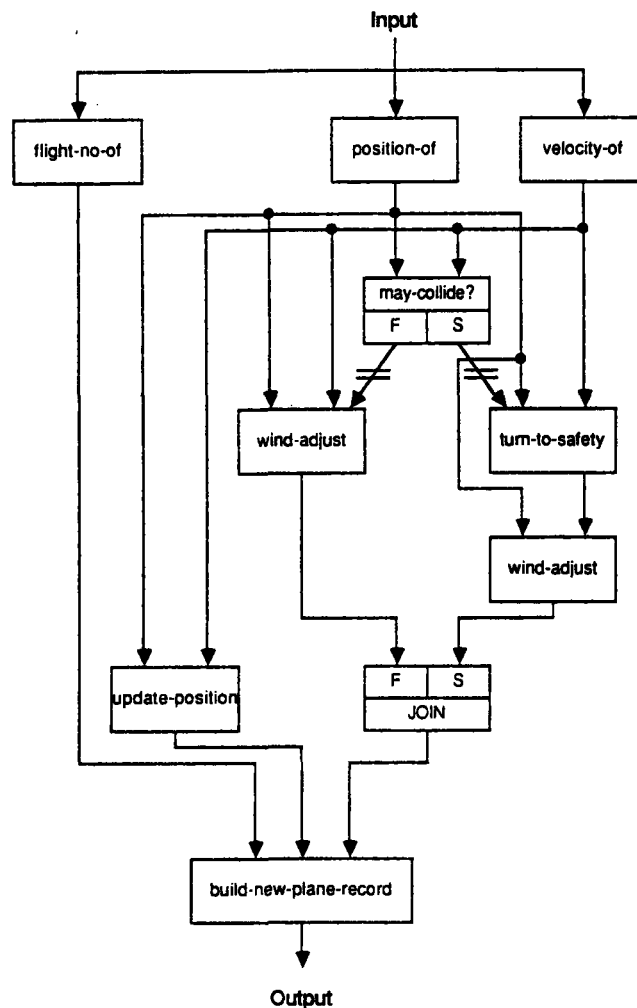


Figure 2: Air Traffic Control Program

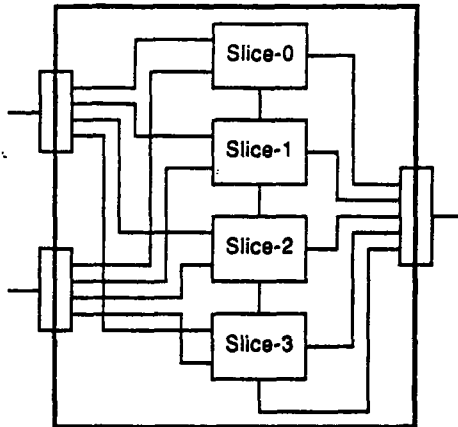


Figure 3: Decomposition of an Add Box

```
(defun update-position (position velocity)
  (vector-sum position
    (vector-scale (delta-t) velocity)))
```

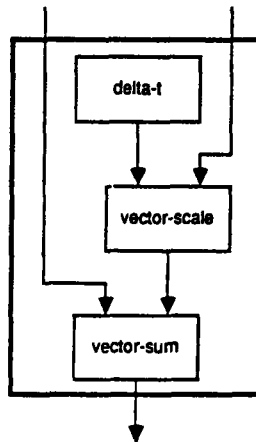


Figure 4: Decomposition of UPDATE-POSITION

while the other two are working. But in software, components are shared. If one wind-adjust box fails, both fail.

This difference between hardware and software will force us to redefine the single-fault assumption. Taken literally, the single-fault assumption tells us that the failure of a single component can be viewed as the failure of a single "box" in the constraint network. But in the presence of shared components, the failure of a single component must be viewed as the failure of *many* boxes in the constraint network.

2.3 The Role of Specifications

In a circuit or program, a discrepancy between specifications and observed behavior indicates the presence of a bug. One way to find a bug is to follow the discrepancies it causes back to the source. Good specifications enable us to detect more discrepancies, thereby allowing us to better localize bugs.

Specifications model the behavior of devices. Since a single specification can rarely describe the full behavior of a device, we often consider collections of partial specifications. A trivial type of partial specification is an enumeration of allowable inputs and outputs for a device. For example, $\{A=3, B=3, C=2, D=2, E=3, F=12, G=12\}$ is a partial specification for the Adder-Multiplier circuit.

1. $Output(Adder) = Input(Adder, A) + Input(Adder, B)$
2. $Input(Adder, A) = Output(Adder) - Input(Adder, B)$
3. $Input(Adder, B) = Output(Adder) - Input(Adder, A)$

Figure 5: Simulation and Inference Rules for an Adder

In hardware, specifications are often given in the form of simulation and inference rules. Simulation rules make “forward” deductions: they allow the outputs of a device to be determined from its inputs. Inference rules make “backward” deductions: they allow one or more inputs of a device to be determined from its outputs and other inputs.

Figure 5 lists the simulation and inference rules for an Adder with inputs A and B. Rule 1 is a simulation rule. Rules 2 and 3 are inference rules. Notice that inference rules do not describe real-world behavior. If one were to put a 10 at the output of the Adder and 4 at input B of the Adder, a 6 would not magically appear at input A.

In software, simulation is easy and inference is hard. Simulation is easy because no rules are needed. If a program works, we can simply execute it to determine its behavior. Inference is hard because programs usually come with less detailed specifications. Without good specifications it’s difficult to come up with inference rules.

There are other ways to describe a device’s behavior besides via simulation and inference rules. One such way to is to describe the dependencies between a device’s inputs and outputs.

Knowing the dependencies between inputs and outputs of a device can be as useful as knowing detailed inference rules. For example, one inference rule for a multiplier states “*If the output is X and one input is A, then the other input should be X/A*”. This rule can be abstracted in terms of dependencies between inputs and outputs as follows: “*If the output is known to be correct, and one input is known to be correct, then the other input should also be correct.*”

Input/output dependencies are useful because they require very little information about the actual function of the device, thus making them easy to provide. For example, the rule for adders is exactly the same as the rule for multipliers, with neither rule mentioning adding or multiplying.

2.4 The Cost of Internal Probing

We probe a circuit or program to determine values at the inputs and outputs of devices. In hardware, probing denotes the physical act of placing an instrument in contact with a wire in the circuit. In software, probing denotes the installation of instrumentation code to monitor control flow and data flow.

There are several considerations in choosing probes. First, if we don't know much about a particular device (i.e., if it's poorly specified), then we might not even be able to conclude whether an observed value is right or wrong. Thus we should avoid probing poorly specified devices.

Second, we should choose probes in accordance with the hierarchical structure of device. Probing begins at the topmost layer of the hierarchy. Once the bug has been localized to particular device in the topmost layer, we decompose that device and begin probing the next layer. Exploiting the hierarchy in this way reduces the number of probes, because the number of components at any level of abstraction is much less than the total number of components in the system.

The final consideration in choosing probes is their cost. Due to physical constraints, probing circuits is often very expensive, if not impossible. Components of a complex circuit may lie within the same physical package, forcing us to observe only the input and output pins of the package. Or a circuit board may be deeply buried within its chassis, preventing us from getting close enough to probe it.

The cost of probing software is more mental than physical, because unlike circuits, programs are relatively free of physical constraint. Physically, editing a low level subroutine is just as easy as editing a top level control loop. But mentally, editing low level routines requires the programmer to consider implementation details he would rather accept on faith.

2.5 Finding Suspects

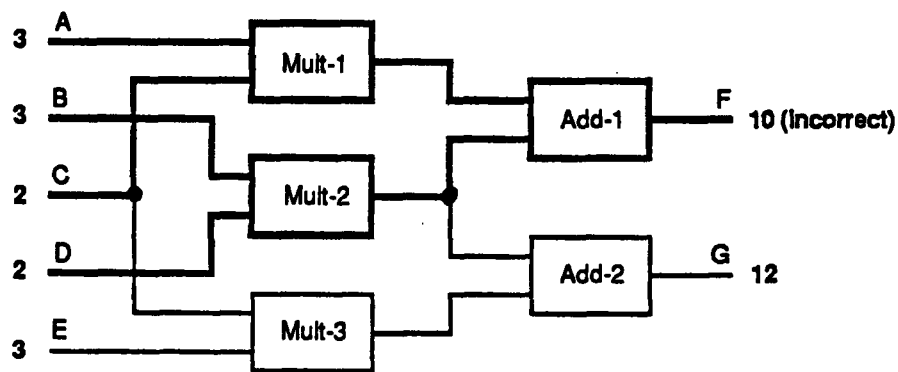


Figure 6: Adder-Multiplier Test

Once we have a device's specifications and have observed discrepancies in its behavior, our task is to determine which component of the device is buggy. First we determine an initial set of suspects from among the device's components. We hope that the initial number of suspects is

significantly less than the total number of components in the device. Then we try exonerate each suspect in turn. This section discusses how we find an initial set of suspects.

In order to find an initial set of suspects we must first determine which components could have contributed to the observed misbehavior. Suspects are found via two simple rules: (1) *A component is suspect if any of its outputs are incorrect*; (2) *A component is a suspect if any of its outputs are connected to another suspect*. The process of finding suspects with these rules is known as *dependency tracing*.

Consider the Adder-Multiplier test shown in Figure 6. The output at F is incorrect: it is 10 when it should be 12. Since F is the output of Add-1, and F is incorrect, Add-1 is a suspect. Since the outputs of Mult-1 and Mult-2 are connected to Add-1, and Add-1 is a suspect, Mult-1 and Mult-2 are suspects. The final set of suspects from this dependency trace is Mult-1, Mult-2 and Add-2.

Our definition of dependency tracing changes when we switch to the domain of software. Consider the Air Traffic Control program shown in Figure 2, which produces only one output. If the output of the program is ever wrong, dependency tracing would uselessly conclude that *all* of the program's components are suspect. Exonerating at least *one* component by dependency tracing would be an improvement. Reasoning about data abstractions in the program will yield the improvement we seek.

The output of the Air Traffic Control program is a *plane-record* constructed by the *build-new-plane-record* function. If any of the inputs to *build-new-plane-record* are incorrect, then the output of the program will be incorrect. Conversely, if the output of the program is incorrect, then *build-new-plane-record* received some incorrect input.

To determine the inputs that were given to *build-new-plane-record* in the creation of a *plane-record* record, we simply apply the accessor functions to the output. Applying accessor functions "spreads" the data structure out into its components.

Figure 7 demonstrates how this idea improves dependency tracing. In this example, the *velocity-of* accessor function relates the program's incorrect output to the third input of *build-new-plane-record*. Dependency tracing from this input yields *wind-adjust*, *turn-to-safety*, *may-collide?*, *position-of* and *vector-of* as suspects.

2.6 Exonerating Provably Innocent Suspects

Dependency tracing leaves us with several suspects. Our next task is try to prove that one or more of the suspects cannot be the true cause of the bug. Control flow analysis can exonerate suspects in programs. *Constraint suspension* [4] can exonerate suspects in circuits as well as programs.

To understand the use of control flow analysis, consider the flow of control in the Air Traffic Control program show in Figure 2. Control passes into the if statement, then into the *may-collide?* predicate. Control then splits, continuing into *turn-to-safety* if *may-collide?* returns TRUE, and into *wind-adjust* if *may-collide?* returns FALSE.

Figure 2 also demonstrates how control flow is represented with plan diagrams. The box labeled with *may-collide?*, F and T represents the if statement in the program. Arrows emanating from F and T represent the the split on control flow based on the outcome of *may-collide?*. Finally, the box labeled with JOIN, F and T represents the synchronization of

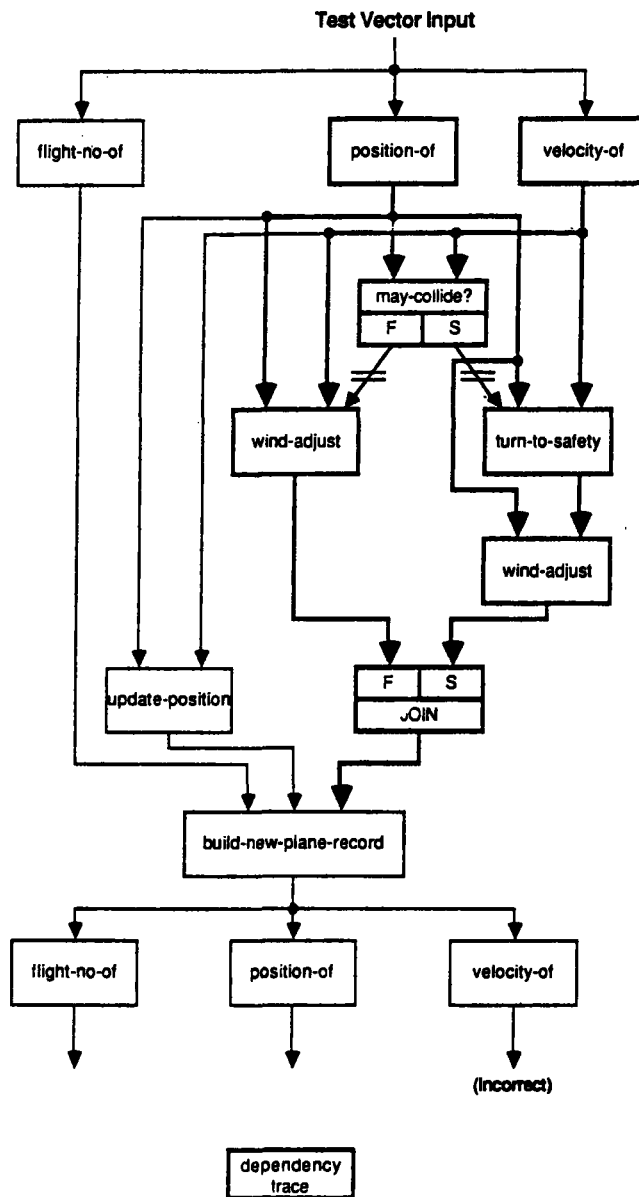


Figure 7: Air Traffic Control Program Test

control flow after the execution of the if.

An analysis of the control flow in the test of the Air Traffic Control Program (Figure 8) allows us to exonerate *turn-to-safety*. Assume that probing tells us that *may-collide?* failed, i.e., the F branch of the split was executed, and the T branch was ignored.

It would be naive to assume that all of the components lying in the ignored T branch went unexecuted. A counterexample to this assumption is *wind-adjust*, which appears in both the T and F branches. *Wind-adjust* *must* have been executed, because the F branch was executed. In other words, appearing in the ignored branch of a split *does not* imply nonexecution.

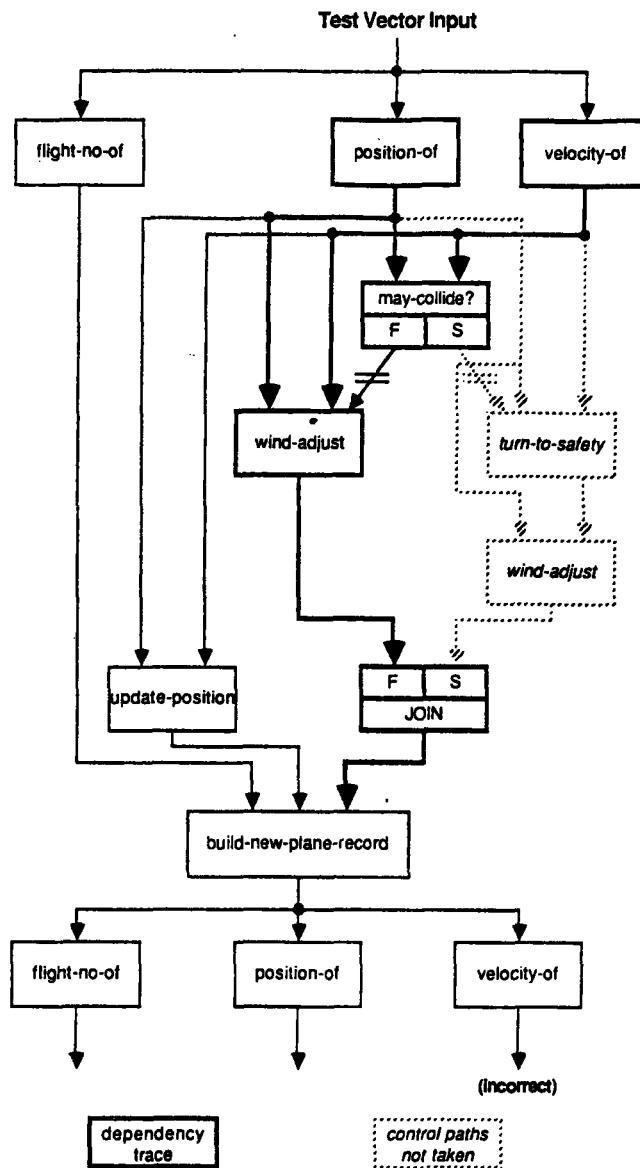


Figure 8: Control Flow in the Air Traffic Control Program Test

Recall that dependency tracing told us that *position-of*, *vector-of*, *may-collide?*, *wind-adjust* and *turn-to-safety* were suspects. But *turn-to-safety* was never executed: it wasn't executed before the split, it doesn't appear in the **F** branch of split, and it wasn't executed after the split. Therefore *turn-to-safety* can be ruled out as a suspect.

We now consider constraint suspension, a technique that is useful in both hardware and software. Constraint suspension is based on the principle that if a device is malfunctioning, then the rules which normally model its behavior no longer apply. If we were to model such a device using constraints, it would have none. So to simulate a malfunctioning device in our network we simply suspend all of the constraints that govern its behavior.

Exonerating a device via constraint suspension proceeds as follows. First, we assume that the device is buggy by suspending its constraints. Next, we place the original test data at the inputs of the network and the observed test results at the outputs of the network. Finally, we run the network's simulation and inference rules.

If running simulation and inference rules leads to a contradiction, then we can exonerate the suspended device by the following argument. The single-fault assumption guarantees that only one device can be broken. Thus each time we suspend the constraints for a device we are implicitly assuming that all other devices are functioning correctly. But if the suspended device is not broken, then the implicit assumption leads to a contradiction: one of the supposedly working devices is actually broken.

Constraint suspension can exonerate **Mult-2** in the Adder-Multiplier test shown in Figure 6. We first assume that **Mult-2** is buggy, so we suspend its constraints. Then the inputs and outputs shown in the diagram are placed at the inputs and outputs of the network. Simulation and inference lead to the following deductions:

1. The output of **Mult-1** is 6, by multiplying A and C.
2. The output of **Mult-2** is 4, by inference on **Add-1**: the first input to **Add-1** is 6, and the output of **Add-1** is 10, so the second input to **Add-1** must be 4. The second input to **Add-1** is connected to the output of **Mult-2**, hence the output of **Mult-2** is also 4.
3. The output of **Mult-3** is 6, by multiplying C and E.
4. The output of **Mult-2** is 6, by inference on **Add-2** (similar to the inference done on **Add-1** in step 2).

The results of (2) and (4) are contradictory, since the output of **Mult-2** cannot be both 4 and 6. **Mult-2** is thereby exonerated.

2.7 Convicting a Suspect

Control flow analysis and constraint suspension will rarely narrow down the set of suspects to a unique element. But we assume that only one component of a circuit or program can be broken. Thus we must determine which of the handful of remaining suspects is the true culprit.

With a complete set of specifications, we can trivially find the culprit: it's the device whose behavior does not agree with its specifications. But given only a partial set of specifications, finding the culprit becomes much harder. This is because only the grossest of errors can be detected by partial specifications.

To illustrate this point, consider a procedure that computes integer factorials. A partial specification on the factorial function is that its result must be greater than zero. Suppose that this hypothetical factorial procedure is buggy in that (**factorial 0**) returns 2 instead of 1. According to the partial specification there is nothing wrong with this result: it is indeed greater than zero.

Convicting a suspect may therefore require user interaction. Suppose some device's behavior agrees with its partial specification. That means that either the device is working or its specification is too vague to detect a problem. The only way to decide is to present the user with the observed inputs and outputs for the device and ask if they seem correct.

We should attempt minimize the number of these user consultations. First, because our goal is to build an automatic debugging system. Second, because the user may be unable to verify the correctness of some input/output pair. And last, because every input/output pair that will be presented to the user has to be acquired at the cost of an additional probe.

One way to minimize the number of user consultations is the *divide and query* approach [24]. Divide and query orders the suspects based on their execution order and dataflow, and then performs a binary search among them to deduce where specifications were first violated.

```
Is (may-collide? #<100 350 18000> #<20 0 -10>) = T correct?
>>> Yes.
Is (wind-adjust #<100 350 18000> #<20 0 -10>) = #<-50 100 0> correct?
>>> No.
```

Figure 9: Divide and Query Applied to the Air Traffic Control Test

Figure 9 illustrates a divide and query scenario in the Air Traffic Control example. The suspects are ordered as follows: **position-of**, **vector-of**, **may-collide?** and **wind-adjust**. **may-collide?** is in the middle of this list, so the user is queried about it first. Prompted with the observed inputs, the user decides that **may-collide?** is indeed working properly.

Knowing the **may-collide?** works allows us to conclude that everything before it in the suspect ordering is innocent. This is because if something before **may-collide?** were broken, then the user would see it through incorrect inputs to **may-collide?**.

We now focus our attention to the suspects that occur after **may-collide?** in the ordering. In our example, the only such suspect is **wind-adjust**; of course there could just as easily have been more than one such suspect. Again the user is queried, but this time he decides the device is indeed buggy. Thus we can finally conclude that **wind-adjust** is the source of the bug.

2.8 Finding Suspects from Multiple Tests or Multiple Faults

The examples just discussed were simplified in order to clarify basic concepts. Specifically, each test produced exactly one discrepancy, and only one test was performed at a time. We now consider multiple discrepancies and multiple tests.

The presence of more than one discrepancy can reduce the size of the initial set of suspects. We need to determine which components can account for all of the discrepancies at once. This is done by intersecting the suspect sets that account for each discrepancy separately.

Performing multiple tests similarly limits the size of suspect sets. We assume that the bug is being caused by one component. This implies that the faulty component will be a suspect in every test. So we intersect the sets of suspects generated from every test.

3 The Debugging Assistant

3.1 Overview

The Debugging Assistant is a prototype of what we hope will eventually become a useful programming tool. Before providing a structured description of the debugging algorithm, we shall briefly summarize its underlying methodology.

The Debugging Assistant does not correct bugs, it only localizes them. Correcting bugs requires an understanding of the relationship between specifications and implementation. The problem of relating specifications to implementation has been ignored to simplify this research.

No heuristic methods are used in the Debugging Assistant. All reasoning is done directly from the structure and behavior of components, i.e., from first principles. Heuristic methods are useful in early steps of debugging, because they directly relate symptoms to bugs and avoid expensive reasoning about the program.

The basic debugging algorithm used by the Debugging Assistant can be applied to programs written in any language. Language independence comes from the use of the plan calculus [12] as a representation for programs.

The Debugging Assistant is simplified by limiting its repertoire of recognizable programs. Programs must be written in functional style, with the exception that side-effecting is permitted is via variable assignments. Loops in a program must be implemented as tail-recursions. This is more a syntactic issue than a restriction, since any loop can be implemented as a tail-recursion.

3.2 Outline of the Debugging Algorithm

1. The procedure being debugged is analyzed to construct a *surface plan* [12, 31], which represents the program as functional boxes with data and control flow constraints.
2. A test case is given to the procedure. Both correct outputs and discrepancies are noted.
3. An initial set of suspects is found via dependency tracing (see above). Sets of suspects from multiple discrepancies or multiple tests are intersected.
4. Components that are provably innocent are exonerated. For each suspect found in step 3:
 - a. By probing splits in control flow, determine if the the suspect was executed. Unexecuted suspects are exonerated.
 - b. If step (a) fails to exonerate the suspect, constraint suspension is applied. If contradictions are found after constraint suspension, the suspect is exonerated.
5. If more than one suspect remains after step 4, try to convict each one in turn:
 - a. For groups of suspects that depend on each other via simple sequential data flow, the user is queried for additional specifications. Some variation of

binary search is applied to minimize the number of queries.

- b. Otherwise, the user is queried for additional specifications in an unspecified order.
6. If one suspect remains, the entire debugging procedure is recursively applied to it, if desired.

4 Related Work

4.1 Overview

Automatic program debugging has been an active area of research in Artificial Intelligence. The design of new debugging systems (this research included) is in part inspired by the successes and failures of old debugging systems.

Debugging systems differ in the way they represent programs. Some systems operate directly on the syntax of the programming language, and are thus deemed to be language dependent. Other systems attempt to model programs in a language independent way, via some graph representation or logical formalism.

Debuggers also differ in the way they reason about programs. Experience-based systems have libraries of heuristics that describe how to find common bugs. Other systems reason from first-principles, using only knowledge about the program to find bugs.

Debugging systems typically perform one or more of the following tasks: program recognition, bug detection, bug localization, bug explanation and bug correction. These tasks are usually done in the order mentioned. Systems which worry more about program testing tend to perform fewer of these tasks (i.e., only recognition and detection). Systems which tutor students must perform all of these tasks.

We can describe the proposed Debugging Assistant in terms of these three aspects. The Debugging Assistant is language independent, by virtue of the plan calculus [12] representation. It reasons about programs from first principles via its use of simple constraints. Finally, the Debugging Assistant localizes bugs and has some ability to explain bugs.

Several criteria are used to evaluate debugging systems. The first is *generality*. An ideal debugging system should be able to detect many types of bugs. It should be able to understand a variety of programs. And it should be able to relate alternate implementations of the same algorithm.

Another criterion is *degree of automation*. The user of the debugging system should be required to do as little work as possible. If the user is an expert programmer, he should not have to answer questions about mundane details of his program. And if the user is a student, he should not have to interpret cryptic error messages.

Some debugging systems claim *cognitive plausibility*. The way a debugging system models programs should somehow parallel the programmer's own mental model. For example, a debugging system that will be used by experts shouldn't model a program at the syntactic level, because experts rarely make deep syntactic errors (i.e., an expert in Pascal will rarely omit a semicolon). But a debugging system that will be used by students must view a program at least partly syntactically, since that's the way students view programs.

The proposed Debugging Assistant can be evaluated by these criteria. It is general, by the argument that any program or bug can be expressed in terms of first principles. It is highly automated, in that control flow analysis and constraint suspension require no user interaction. And it is cognitively plausible, because everyone must resort to first principles when experience is of no help.

4.2 Tutoring Systems

One application of automatic program debugging is the tutoring of novice programmers. Tutoring systems are usually experience-based: they maintain a library of algorithm descriptions which serve as templates for correct student programs. The tutoring system will compare a student's code to the appropriate algorithm description, transforming one or the other to account for minor implementation differences.

If a student's program cannot be matched to the algorithm description, experience-based bug detection is invoked. One by one, a collection of bug experts, each knowing the symptoms and cure for a specific bug, examines the code. When able, an expert modifies the buggy code to correct the bug and allow matching to continue.

When evaluating tutoring systems, we stress the importance that the systems provide good explanations. A good explanation describes the cause of a bug rather than its symptom. If a student understands how a bug arises, he or she can learn how to program defensively and prevent the bug from appearing again.

In evaluating tutoring systems we emphasize the need for cognitive plausibility. A tutoring system is not only debugging programs, it is debugging the mind of the student. Any bug in a program can be traced to a specific misunderstanding in the mind of the student. A good debugging model makes this relationship explicit.

In Ruth's system [21], *Program Generation Models (PGM's)* describe algorithms by modeling the decisions made in writing a program. A PGM is like a context free grammar; Where context-free grammars derive valid strings in a language; PGM's derive valid implementations of an algorithm. A recursive descent parser called the *Action List Matcher (ALM)* attempts to match a program to a PGM.

When the ALM is unable to parse a section of code, a bug has been found. Some bugs, such as loops that repeat the wrong number of times, require only minor changes in the source code to be corrected. These bugs are heuristically detected and corrected, thereby allowing the parse to continue. Other bugs, such as missing control structures, can only be corrected by major changes to the source code. These bugs indicate either that the wrong PGM is being matched with the program, or that the program is grossly incorrect.

PGM's are not guaranteed to represent all possible implementations of a given algorithm. If a student has a syntactically mangled implementation that happens to work, Ruth's system might consider the program incorrect. And if the student devises some clever new implementation of an algorithm, the PGM might not be able to derive it.

Adam and Laurent's LAURA [1] represents algorithms as *program models*. A program model is a supposedly correct implementation of an algorithm. Program models are written by the teacher in a traditional iterative language (FORTRAN).

LAURA converts the student's program and the program model into labeled control-flow graphs. During this process a variety of transformations are systematically applied to canonicalize graph structure. LAURA then compares the two graphs, applying additional transformations in an attempt to make the graphs as similar as possible. Finally, any remaining differences are diagnosed from a set of known errors.

LAURA does not suggest corrections for errors, nor does it refer to syntactic elements of the program in its error messages. Instead, it presents the program model along with an annotated transformed version of the student's program. Examples of the annotations LAURA provides are "Line 15 in program 1 is unidentifiable" or "Different conditions on the arcs coming from lines 10 and 109."

The actual utility of presenting the student with annotated rewritten programs is questionable. An inexperienced student may not be able to understand why the rewritten version of his program is more correct than the original. And the vague annotations LAURA provides do not tell enough about what is actually wrong with the code. The student would learn more from a message like "Bad initialization of variable N in line 3" than he would from "Undefined instruction in line 3."

Program models in LAURA share the same shortcomings as PGM's in Ruth's system. An implementation of an algorithm may be correct even though its structure differs significantly from the program model.

Murray's TALUS [11] combines heuristic and formal methods. Heuristic methods are used to recognize algorithms, to guess at the possible locations of bugs and to suggest corrections for bugs. Formal methods are used to verify the equivalence of program fragments, to detect bugs and to prove or disprove heuristic conjectures.

TALUS views all programs, either student or teacher written, as collections of functions. Functions have abstract features such as recursion type and termination conditions. The measure of similarity between two functions is the number of abstract features they share. The measure of similarity between two programs is a weighted sum of the similarities of their component functions.

The first stage of debugging in TALUS is algorithm recognition. TALUS performs a best first search through all known algorithms to find the one algorithm that is most similar to the student's solution. Transformations are applied to facilitate matching with algorithms that have several functional decompositions.

The second stage of debugging is bug detection. In this stage, functions are represented as binary trees, with internal nodes representing conditional tests and leaf nodes representing function terminations or recursions. The set of conditions which must be true to reach a given leaf node defines a test case for that node. TALUS supplies each test case to both the student's solution and the matched algorithm. If the resulting returned values do not agree, a bug has been found.

The third and final stage of debugging in TALUS is bug correction. Top level expressions in the student's code fragment are replaced with their counterparts in the teacher's algorithm. When the two code fragments are found to be functionally equivalent, the bug has been completely corrected.

Representing an algorithm as a collection of abstract properties has several advantages. Algorithms are matched on the basis of abstract nonsyntactic features, so syntactically unconventional implementations will always be recognized. The properties which describe programs are language independent; with the appropriate parsers, algorithms and solutions can be written in any programming language. Bug descriptions drawn from abstract properties can replace a symptom with its cause (i.e., a message like "The loop variable X has been incorrectly

initialized” describes a symptom, whereas “The DO loop over variable X repeats 1 time too many” describes the cause of the symptom).

Johnson and Soloway’s PROUST [8] debugs programs by reconstructing the goals of the student and identifying the elements of the program that were meant to realize the goals. This process is claimed to correspond to the actual thoughts of the student as he or she writes a program.

PROUST uses *programming plans* to represent common implementation fragments, both correct and buggy. For example, the “counter plan” describes the code where a variable is assigned an initial value and then incremented within the body of a loop. Programming plans are founded on the theory that expert programmers reason in terms of familiar algorithmic fragments, as opposed to primitive language constructs.

A programming task can be broken down into subtasks. A *goal decomposition* of a program describes the hierarchical structure of its subtasks, how its subtasks interact, and the mapping of subtask goals to the plans which implement them. PROUST relates programs to goals by matching plans from the goal decomposition to the program’s code.

A problem description in PROUST can give rise to many correct and incorrect implementations. The initial description of the problem may have several goal decompositions, and each subtask in a goal decomposition may be implemented by several different plans.

PROUST avoids searching through all implementations of a task by using heuristics that describe which plans and goals will occur together. Thus goals are decomposed at the same time as plans are analyzed. As PROUST begins to understand a program, it establishes expectations to confirm its current line of reasoning. When an expectation fails, PROUST tries an alternate interpretation for the program.

PROUST is most useful as a tutoring tool. By attempting to capture the cognitive processes in program synthesis, it can assist a misguided student by appealing to his or her deeper understanding of program design. This is a feature missing in LAURA or TALUS, which simply present the bug in the code and suggest a repair.

4.3 Debugging Systems

Daniel Shapiro’s Sniffer [23] uses expert knowledge about programming to understand specific errors. Sniffer recognizes programs by identifying familiar algorithmic fragments, or *programming cliches* [12, 16, 31] in the code. Knowledge about bugs is encoded in bug experts, which generate detailed reports about errors.

A debugging session in Sniffer proceeds as follows. The user asks Sniffer to execute his program. As the program runs, Sniffer constructs an execution history containing information about when and where variables were modified, and what paths of control flow were taken.

The user interrupts execution at the first sign of trouble. He uses the *time rover* to search the program’s execution history for bug symptoms and to localize the bug to a particular section of code. Once the location of the bug has been found, the programmer asks the *sniffer system* for a report.

The sniffer system performs two functions. First, it employs a *cliche finder* to recognize the

familiar parts of the buggy code. Then bug experts are invoked to determine the exact nature of the bug. Bug experts use the time rover to verify symptoms for the bugs they specialize in.

Finally, Sniffer produces a detailed report about the bug. This report summarizes the error, analyzes the intended function of the code, and discusses how the bug manifested itself at runtime.

An advantage of Sniffer is its well defined modularity. In theory, one could easily augment the knowledge base of either the cliché finder or the sniffer system to suit any domain of possible bugs.

Because Sniffer is not given any specification information, it can neither detect nor localize bugs. This places unreasonable demands on the user, especially in large software systems. Bugs can manifest themselves in subtle ways in large systems, making their detection difficult. The number of components in a large system complicates the task of tracing a bug to its source.

Lukey's PUDSY [10] understands a program by building a description of the program. These descriptions can be compared to specifications to find bugs. Bugs occur where descriptions disagree with specifications.

Building a program description in PUDSY proceeds as follows. First, the program is grouped into *chunks*. A chunk is a schema for common computations. A common type of chunk is a loop that finds an array's maximum element. PUDSY determines the dataflow in and out of each chunk, and the dataflow between chunks.

Next, PUDSY looks for *debugging clues* by using constraints on what "rational" programs look like. One such constraint is that a variable rarely appears in the left hand side of two consecutive assignment statements. Another constraint is that variable names are meaningful: a variable named *min* usually finds a minimum element. Violations of these constraints are usually noted for later use, but in some instances they can be used to immediately debug a section of code.

Describing a program in PUDSY is viewed as a stepwise process, where each step performs some transformation on the current description. An initial description of a chunk is made by trying to recognize it as an instance of a known schema. Every schema that can be recognized by PUDSY comes with a logical assertion that describes it. Assertions are combined by reasoning about program control and data flow. For example, a chunk that appears in the body of a loop can be quantified over the loop variable.

If the final program description does not agree with its specification, a bug has been found. PUDSY applies *backtracing* to determine the source of the bug. In backtracing, the inverse of each description-building transformation is applied to the program's specification. For example, if a transformation quantified an assertion in the description, back tracing would remove the equivalent quantification from the specification. In this way PUDSY can find the first point where descriptions and specifications disagree.

PUDSY's methodology for finding bugs is useful and reliable. Comparing complete specifications to descriptions will always find a bug if there is one. And looking for discourse clues in variable names is good way to detect low level differences between what the programmer meant to do and what he did by mistake.

Ehud Shapiro's system [24] debugs Prolog programs from first principles. Shapiro's system

comes closest to this research in its use of first-principles reasoning. Three types of bugs are considered by this system: *termination with incorrect output*, when the output value of a deterministic procedure is incorrect; *finite failure*, when none of the outputs of a nondeterministic procedure are correct; and *nontermination*, when the program enters an infinite loop.

A debugging session in Shapiro's system consists of a question and answer session with the user. If some input causes a program to terminate with incorrect output, the system will selectively ask the user about the correctness of intermediate results of the computation. An approach termed *divide and query* performs a binary search on the steps of the computation to quickly focus in on the source of the bug.

Debugging a finite failure condition proceeds in a similar way. In this case, since the buggy procedure is nondeterministic, the debugger asks the user to supply *all* known solutions to intermediate results (making what is called an *existential query*).

Nontermination is debugged in several ways. First, the program can be run with bounds on space or time, on the assumption that exceeding these bounds implies that the program does not terminate. Also, *well founded orderings* can be defined on a procedure. An example of a well founded ordering is that consecutive calls to a divide and conquer procedure have decreasing parameter size.

Shapiro's work goes beyond debugging. He proposes a method for the inductive learning of programs (beyond the scope of this paper), and then demonstrates how the inductive learner can be applied to the correction of bugs. The underlying idea is that learning a program is incremental. At some point in time we have a partially learned version of the program. As various input/output pairs come in to further describe the program's behavior, the partial definition of the program is modified slightly. To debug a program, one could use the input/output pair that caused the bug to manifest itself as a negative example. Similarly, one could supply correct input/output pairs to buggy programs as positive examples.

Shapiro's system is able to localize broad classes of bugs by reasoning from first principles. A couple of minor details in the methodology used in localizing the bugs are suspect, however. The first problem is that the user is treated as an oracle that can answer yes or no questions about the desired behavior of the program. But the answers to these questions should be found by consulting the specifications for the program. The user could still be consulted, but only if the specifications are too unclear or unwieldy to get a straight answer.

Another shortcoming is that there is no way for Shapiro's system to explain to the user why it is asking a particular question. In reading a debugging session with the divide and query approach, one sees that the questions asked seem relatively unrelated. The debugger should explain why it chose to ask the user the seemingly random handful of questions it did.

Gupta and Seviora's Message Trace Analyzer [5] uses an expert systems approach for debugging real time processes. Each process is modeled by a finite state machine that interacts with other processes by sending messages. The system constructs a structured model of interprocess communication called the *context tree*. The construction of the context tree is done through a multilevel subgoaling process. Components of the context tree are tested for failure by heuristic rules and state machine simulation.

The approach taken in the Message Trace Analyzer does not lend itself to the general

debugging of traditional serial software systems. Debugging based solely on interprocess communication is akin to a pure I/O based debugging approach. In a complex software system, simple I/O discrepancies could have many equally valid explanations. One needs to understand the internal behavior of a program (or at least how it can be decomposed into simpler parts) in order to debug it.

One promising feature in the Message Trace Analyzer is its separation of general knowledge of real time systems from specific domain knowledge (the domain being telephone switching systems). This parallels the need for a software debugger to separate first principles programming knowledge from the knowledge of specific algorithms. A debugger which can maintain both types of knowledge and can intelligently decide to use one or the other would be quite useful indeed.

Harandi's Knowledge Based Programming Assistant [7] is another expert-systems approach. In this system, heuristic information is used to find many compile time and run time errors with well-defined symptoms. These heuristics are specified as situation/action pairs. The situation specifies bug symptoms and program information, and the action describes probable causes for the error and possible cures.

The apparent intent of [7] is to present a description of the knowledge base structure and inference system operation. Unfortunately, none of the actual rules for debugging are presented in the work.

4.4 Other Work

Waters [33] observes that two approaches have been traditionally used in the verification and debugging process, testing and inspection. Both approaches have problems when a large system must be dealt with. The utility of testing is limited by the imagination of the programmer who designs the tests. If the programmer cannot envision some unexpected error condition, he will not devise a test for it. The power of inspection is limited by the complexity of subprogram interactions in a large system. A programmer that inspects code to verify its correctness might not have the insight to consider the interaction of two seemingly unrelated subroutines.

Constraint modeling is a third way to verify and debug programs. The program is modeled as a network of constraints. The choice of what aspects of the program to model and what constraints to use is left up to the programmer. By performing constraint propagation on this network, bugs can be found that might not be found using testing or inspection. Waters concludes that the three approaches of testing, inspection and constraint modeling are mutually orthogonal, and are best used together in system verification.

Constraint modeling is the primary strategy used by the Debugging Assistant and by most hardware troubleshooters. Constraints are given by the specifications of the components of the program and their interconnections. Bugs are found by detecting contradictions (discrepancies) in this constraint network. Testing is used as a secondary strategy, as a method for determining an initial set of contradictions.

Chapman's Program Testing Assistant [2] helps programmers develop and maintain program test cases. The programmer tests functions in his program by specifying an expression

to execute on some test data, along with correct results and success criteria. Each test case is associated with the set of functions it verifies through a set of abstract features. If any of those functions change, the test is re-run. If a success criterion is not met the programmer is warned of the error.

Wills' program recognizer [34] applies flow-graph parsing to the recognition of programs as plans in the plan calculus. A program is first transformed into a surface plan by control and data flow analysis. Surface plans represent programs in terms of functional boxes, data flow, control flow and constraints (the Debugging Assistant represents programs as surface plans). This surface plan is then translated into an extended flow graph (a type of labeled, acyclic, directed graph) to better facilitate subsequent matching. Flow graphs are parsed against a library of common structures to determine familiar program fragments. During the parse the original graph may be transformed in order to eliminate constraint violations.

Program recognition has always been considered an integral part of debugging. Wills' program recognizer factors this task out of the debugging process, allowing future research to concentrate more on bug localization and correction.

Levitin [9] explores the meaning and uses of errors in programming. A roughly day-long coding assignment in CLU (a strongly typed high-level language) was given to several volunteers. Versions of the program files were examined after the completion of the project to determine the quantity and nature of bugs encountered. Bugs were classified by such names as *missing guard*, *missing declaration*, and *malformed update*. Levitin concludes from this experiment that a general method for describing bugs is needed, one that works equally well for any programming task.

The method for describing bugs proposed by Levitin describes a bug as a vector in a space of categories. Each category has a metric associated with it that describes how the bug relates to that category. The categories chosen are *severity of error*, how serious the error is to the development process; *locus of error*, at what level of thought process the programmer erred; and *intent of error*, the realization of the programmer that a mistake was being made.

The metrics proposed vary depending on the category. Locus of error is measured across the spectrum from specification to implementation. Severity of error can be measured in amount of code changed, amount of time taken, or combinations of these and similar metrics. Intent of error is quantified based on the stage of the design process where the programmer decided to ignore some assumption about the program, and when the programmer realized the exact nature of the assumptions being violated.

References

1. Adam, Anne and Laurent, Jean-Pierre. "LAURA, a System to Debug Student Programs". *Artificial Intelligence* 15, 1 (November 1980), 75-122.
2. Chapman, David. "A Program Testing Assistant". *Communications of the ACM* 25, 9 (September 1982).
3. Cyphers, D. Scott. Automated Program Description. (MIT-AI Working Paper 237).
4. Davis, Randall. Diagnostic Reasoning Based on Structure and Behavior. AI Memo 739, MIT AI Laboratory, June, 1984.
5. Gupta, N. K. and Seviara, R. E. An Expert System Approach to Real Time System Debugging. Proceedings of the First Conference on Artificial Intelligence Applications, December, 1985.
6. Hamscher, Walter and Davis, Randall. Issues in Model Based Troubleshooting. AI Memo 893, MIT AI Laboratory, March, 1987.
7. Harandi, Mehdi T. Knowledge-Based Program Debugging: A Heuristic Model. Proceedings of SOFTFAIR, July, 1983.
8. Johnson, W. Lewis and Soloway, Elliot. PROUST: Knowledge-Based Program Understanding. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters, Eds., Morgan Kaufmann, 1986, pp. 443-451.
9. Levitin, Samuel M. Toward a Richer Language for Describing Software Errors. (MIT-AI Working Paper 270).
10. Lukey, F. J. "Understanding and Debugging Programs". *International Journal on Man-Machine Studies* 14 (February 1980), 189-202.
11. Murray, William R. Heuristic and Formal Methods in Automatic Program Debugging. Proceedings of the IJCAI, August, 1985.
12. Rich, Charles. Inspection Methods in Programming (PhD Thesis). AI-TR 604, MIT AI Laboratory, June, 1981.
13. Rich, Charles. A Formal Representation for Plans in the Programmer's Apprentice. Proceedings of the IJCAI, August, 1981.
14. Rich, Charles. Knowledge Representation Languages and the Predicate Calculus: How to Have Your Cake and Eat It Too. Proceedings of the AAAI, August, 1982.
15. Rich, Charles. The Layered Architecture of a System for Reasoning about Programs. Proceedings of the IJCAI, August, 1985.
16. Rich, Charles and Waters, Richard C. Abstraction, Inspection and Debugging in Programming. AI Memo 634, MIT AI Laboratory, June, 1981.
17. Rich, Charles and Waters, Richard C. The Disciplined Use of Simplifying Assumptions. (MIT-AI Working Paper 220).
18. Rich, C. and Waters, Richard C. Toward a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications. AI Memo 907, MIT AI Laboratory, July, 1986.

19. Rich, Charles and Waters, Richard C. A Scenario Illustrating a Proposed Program Design Apprentice. AI Memo 933A, MIT AI Laboratory, January, 1987.
20. Rich, Charles and Waters, Richard C. Formalizing Reusable Software Components in the Programmer's Apprentice. AI Memo 954, MIT AI Laboratory, February, 1987.
21. Ruth, Gregory R. Intelligent Program Analysis. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters, Eds., Morgan Kaufmann, 1986, pp. 431-441.
22. Seviara, Rudolph E. "Knowledge-Based Program Debugging Systems". *IEEE Software Magazine* 20, 5 (May 1987), 20-31.
23. Shapiro, Daniel G. Sniffer: A System that Understands Bugs (MS Thesis). AI Memo 638, MIT AI Laboratory, June, 1981.
24. Shapiro, Ehud Y. Algorithmic Program Debugging (PhD Thesis). Yale RR 237, Yale University, Department of Computer Science, May, 1982.
25. Soloway, Elliot and Ehrlich, Kate. Empirical Studies of Programming Knowledge. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters, Eds., Morgan Kaufmann, 1986, pp. 507-521.
26. Steele, Guy Lewis, Jr. The Definition and Implementation of a Computer Programming Language Based on Constraints (PhD Thesis). AI-TR 595, MIT AI Laboratory, August, 1980.
27. Steele, Guy Lewis, Jr.. *Common LISP*. Digital Press, 1984.
28. Sussman, Gerald Jay and Steele, Guy Lewis, Jr. CONSTRAINTS, A Language for Expressing Almost-Hierarchical Descriptions. AI Memo 502A, MIT AI Laboratory, August, 1981.
29. Tan, Yang Meng. ACE: A Cliche-based Program Structure Editor. (MIT-AI Working Paper 294).
30. Waters, Richard C. "A Method for Analyzing Loop Programs". *IEEE Transactions on Software Engineering SE-5*, 3 (May 1979).
31. Waters, Richard C. KBEmacs: A Step Towards the Programmer's Apprentice. AI-TR 753, MIT AI Laboratory, May, 1985.
32. Waters, Richard C. Program Translation Via Abstraction and Reimplementation. AI Memo 949, MIT AI Laboratory, December, 1986.
33. Waters, Richard C. System Validation via Constraint Modeling.
34. Wills, Linda M. Automated Program Recognition (MS Thesis). AI-TR 904, MIT AI Laboratory, February, 1987.
35. Zelinka, Linda M. An Empirical Study of Program Modification Histories. (MIT-AI Working Paper 240).